

Weak Visibility Queries of Line Segments in Simple Polygons

Danny Z. Chen^{1*} and Haitao Wang^{2**}

¹ Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN 46556, USA
dchen@cse.nd.edu

² Department of Computer Science
Utah State University, Logan, UT 84322, USA
haitao.wang@usu.edu

Abstract. Given a simple polygon P in the plane, we present new algorithms and data structures for computing the weak visibility polygon from any query line segment in P . We build a data structure in $O(n)$ time and $O(n)$ space that can compute the visibility polygon for any query line segment s in $O(k \log n)$ time, where k is the size of the visibility polygon of s and n is the number of vertices of P . Alternatively, we build a data structure in $O(n^3)$ time and $O(n^3)$ space that can compute the visibility polygon for any query line segment in $O(k + \log n)$ time.

1 Introduction

Given a simple polygon \mathcal{P} of n vertices in the plane, two points in \mathcal{P} are *visible* to each other if the line segment joining them lies in \mathcal{P} . For a line segment s in \mathcal{P} , a point p is *weakly visible* (or *visible* for short) to s if s has at least one point that is visible to p . The *weak visibility polygon* (or *visibility polygon* for short) of s , denoted by $Vis(s)$, is the set of all points in \mathcal{P} that are visible to s . The *weak visibility query problem* is to build a data structure for \mathcal{P} such that $Vis(s)$ can be computed efficiently for any query line segment s in \mathcal{P} .

This problem has been studied before. Bose *et al.* [2] built a data structure of $O(n^3)$ size in $O(n^3 \log n)$ time that can compute $Vis(s)$ in $O(k \log n)$ time for any query, where k is the size of $Vis(s)$. Throughout this paper, we always let k denote the size of $Vis(s)$ for any query line segment s . Bygi and Ghodsi [3] gave an improved data structure with the same size and preprocessing time as that in [2] but its query time is $O(k + \log n)$. Aronov *et al.* [1] proposed a smaller data structure of $O(n^2)$ size with $O(n^2 \log n)$ preprocessing time and $O(k \log^2 n)$ query time. Table 1 gives a summary. If the problem is to compute $Vis(s)$ for a single segment (not queries), then there is an $O(n)$ time algorithm [11].

* The research of D.Z. Chen was supported in part by NSF under Grants CCF-0916606 and CCF-1217906.

** Corresponding author.

Table 1. A summary of the data structures. The value k is the size of $Vis(s)$ for any query segment s .

Data Structure	Preprocessing Time	Size	Query Time
[2]	$O(n^3 \log n)$	$O(n^3)$	$O(k \log n)$
[3]	$O(n^3 \log n)$	$O(n^3)$	$O(k + \log n)$
[1]	$O(n^2 \log n)$	$O(n^2)$	$O(k \log^2 n)$
Our Result 1	$O(n)$	$O(n)$	$O(k \log n)$
Our Result 2	$O(n^3)$	$O(n^3)$	$O(k + \log n)$

1.1 Our Contributions

In this paper, we present two new data structures whose performances are also given in Table 1. Our first data structure, which is built in $O(n)$ time and $O(n)$ space, can compute $Vis(s)$ in $O(k \log n)$ time for any query segment s . Comparing with the data structure in [1], our data structure reduces the query time by a logarithmic factor and uses much less preprocessing time and space.

The preprocessing time and size of our second data structure are both $O(n^3)$, and each query takes $O(k + \log n)$ time. Comparing with the result in [3], our data structure has less preprocessing time. In addition, our solution, which is based on the approach in [2], is much simpler than that in [3]. Further, our techniques explore many geometric observations on the problem that may be useful elsewhere. For example, we prove a tight combinatorial bound for the “zone” in a line segment arrangement contained in a simple polygon, as follows, which is interesting in its own right.

Let S be a set of line segments in a simple polygon \mathcal{P} such that both endpoints of each segment of S are on $\partial\mathcal{P}$ (i.e., the boundary of \mathcal{P}). Let \mathcal{A} be the arrangement formed by the segments in S and the edges of $\partial\mathcal{P}$. For any line segment s in \mathcal{P} (the endpoints of s need not be on $\partial\mathcal{P}$), the *zone* of s , denoted by $Z(s)$, is defined to be the set of all faces of \mathcal{A} that s intersects. For each edge of any face in \mathcal{A} , it either lies on a segment of S or lies on $\partial\mathcal{P}$. Let Λ be the number of edges of the faces in $Z(s)$ each of which lies on a segment of S . We want to find a good upper bound for Λ . By using the zone theorem for the general line segment arrangement [8], we can easily obtain $\Lambda = O(|S|\alpha(|S|))$, where $\alpha(\cdot)$ is the functional inverse of Ackermann’s function [13]. In this paper, we prove a tight bound $\Lambda = O(m)$, where $m \leq |S|$ is the number of segments in S each of which contains at least one edge of the faces in $Z(s)$. An immediate application of this result is that we obtain an efficient query algorithm for our second data structure. Since combinatorial bounds on arrangements are fundamental, this result may find other applications as well.

The rest of this paper is organized as follows. In Section 2, we review some geometric structures and a query algorithmic scheme that will be used by the query algorithms of both our data structures. We will also give a “ray-rotating” data structure in Section 2, which is needed by our first data structure in Section 3. In Sections 3 and 4, we present our first and second data structures, respectively. As a by-product of our second data structure, the combinatorial bound

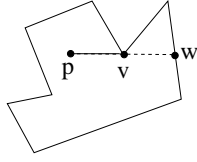


Fig. 1. Illustrating a window \overline{vw} of p .

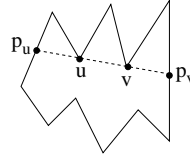


Fig. 2. Illustrating the two critical constraints $\overline{vp_v}$ and $\overline{up_u}$ defined by the two mutually visible vertices u and v .

of the zone mentioned above is also given in Section 4. Section 5 concludes the paper.

2 Preliminaries

In this section, we review some geometric structures and discuss an algorithmic scheme that will be used by the query algorithms of both our data structures given in Sections 3 and 4. We will also give a “ray-rotating” data structure in Section 2.1, which is needed by our first data structure in Section 3.

For simplicity of discussion, we assume no three vertices of \mathcal{P} are collinear; we also assume for any query segment s , s is not collinear with any vertex of \mathcal{P} and each endpoint of s is not collinear with any two vertices of \mathcal{P} . As in [1,2], our approaches can be easily extended to the general situation.

Denote by $\partial\mathcal{P}$ the boundary of \mathcal{P} . The visibility graph of \mathcal{P} is a graph whose vertex set consists of all vertices of \mathcal{P} and whose edge set consists of edges defined by all visible pairs of vertices of \mathcal{P} . Here, two adjacent vertices on $\partial\mathcal{P}$ are considered visible to each other. In this paper, we always use K to denote the size of the visibility graph of \mathcal{P} . Note that $K = O(n^2)$ and $K = \Omega(n)$. The visibility graph can be computed in $O(K)$ time [14].

We introduce the *visibility decomposition* of \mathcal{P} [1,2]. Consider a point p in \mathcal{P} and a vertex v of \mathcal{P} . Suppose the line segment \overline{pv} is in \mathcal{P} , i.e., p is visible to v . We extend \overline{pv} along the direction from p to v and suppose we stay inside \mathcal{P} (when this happens, v must be a reflex vertex). Let w be the point on the boundary of \mathcal{P} that is hit first by our above extension of \overline{pv} (e.g., see Fig. 1). We call the line segment \overline{vw} the *window* of p . The point p is called the *defining point* of the window and the vertex v is called the *anchor vertex* of the window. It is well known that the boundary of the visibility polygon of the point p consists of parts of $\partial\mathcal{P}$ and the windows of p [1,2]. If the point p is a vertex of \mathcal{P} , then the window \overline{vw} is called a *critical constraint* of \mathcal{P} and p is called the *defining vertex* of the critical constraint. For example, in Fig. 2, the two critical constraints $\overline{up_u}$ and $\overline{vp_v}$ are both defined by u and v ; for $\overline{up_u}$, its anchor vertex is u and its defining vertex is v , and for $\overline{vp_v}$, its anchor vertex is v and defining vertex is u . It is easy to see that the total number of critical constraints is $O(K)$ because each critical constraint corresponds to a visible vertex pair of \mathcal{P} and a visible vertex pair corresponds to at most two critical constraints.

As in [1,2], we represent the visibility polygon $Vis(s)$ of a segment s by a cyclic list of the vertices and edges of \mathcal{P} in the order in which they appear on the boundary of $Vis(s)$, and we call such a list the *combinatorial representation* of $Vis(s)$ [1]. With the combinatorial representation, $Vis(s)$ can be explicitly determined in linear time in terms of the size of $Vis(s)$. Our query algorithms given later always report the combinatorial representation of $Vis(s)$.

The critical constraints of \mathcal{P} partition \mathcal{P} into cells, called the *visibility decomposition* of \mathcal{P} and denoted by $VD(\mathcal{P})$. The visibility decomposition $VD(\mathcal{P})$ has a property that for any two points p and q in the same cell of $VD(\mathcal{P})$, the two visibility polygons $Vis(p)$ and $Vis(q)$ have the same combinatorial representation. Also, the combinatorial representations of the visibility polygons of two adjacent cells in $VD(\mathcal{P})$ have only $O(1)$ differences. The visibility decomposition has been used for computing visibility polygons of query points (not line segments) [1,2].

Consider a query segment s in \mathcal{P} . In the following, we discuss an algorithmic scheme for computing $Vis(s)$. Denote by a and b the two endpoints of s . Suppose we move a point p on s from a to b . We want to capture the combinatorial representation changes of $Vis(p)$ of the point p during its movement. Initially, p is at a and we have $Vis(p) = Vis(a)$. As p moves, the combinatorial representation of $Vis(p)$ changes if and only if p crosses a critical constraint of \mathcal{P} [1,2]. $Vis(s)$ is the union of all such visibility polygons as p moves from a to b . Therefore, to compute $Vis(s)$, as in [1,2], we use the following approach. Initially, let $Vis(s) = Vis(p) = Vis(a)$. As p moves from a to b , when p crosses a critical constraint, either p sees one more vertex/edge, or p sees one less vertex/edge. If p sees one more vertex/edge, then we update $Vis(s)$ in constant time by inserting the new vertex/edge to the appropriate position of the combinatorial representation of $Vis(s)$. Otherwise, we do nothing (because even though a vertex/edge is not visible to p any more, it is visible to s and thus should be kept; refer to [2] for details).

The above algorithm has two remaining issues. The first one is how to compute $Vis(a)$ of the point a . The second issue is how to determine the next critical constraint that will be crossed by p as p moves. Each of our two data structures given in Sections 3 and 4 does some preprocessing such that the corresponding query algorithm can resolve the above two issues efficiently.

2.1 The Ray-Rotating Queries

Our first data structure in Section 3 needs the following *ray-rotating* queries. Given any ray ρ whose origin z is in \mathcal{P} , the ray-rotating query asks for the first vertex of \mathcal{P} visible to z that will be hit by ρ when we rotate ρ clockwise (or counterclockwise) around z (e.g., see Fig. 3). By making use of the ray-shooting data structures [5,6,11,15] and the two-point shortest path query data structure [12], we obtain the following result.

Lemma 1. *A data structure can be built in $O(n)$ time and $O(n)$ space such that each ray-rotating query can be answered in $O(\log n)$ time.*

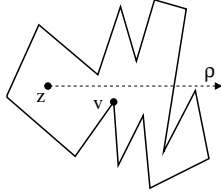


Fig. 3. Illustrating the ray-rotating query for ρ : The vertex v is the first visible vertex to z that will be hit by ρ if we rotate ρ clockwise around z .

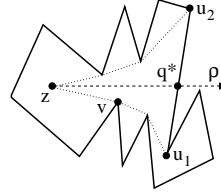


Fig. 4. Illustrating the proof for Lemma 1: The two dotted paths are shortest paths from z to u_1 and u_2 , respectively.

Proof: Consider any ray ρ whose origin z is in \mathcal{P} . Without loss of generality, assume ρ is horizontally rightwards. Let v^* be the sought vertex for the ray-rotating query on ρ , i.e., v^* is the first vertex of \mathcal{P} visible to z that will be hit by ρ when we rotate ρ clockwise around z (the case of counterclockwise rotation can be done similarly).

In the preprocessing, we compute a ray-shooting data structure [5,6,11,15] in $O(n)$ time and space, such that given any ray with the origin in \mathcal{P} , the first point on the boundary of \mathcal{P} hit by the ray can be found in $O(\log n)$ time. We also compute a two-point shortest path query data structure [12] in $O(n)$ time and space, such that given any two points p and q in \mathcal{P} , the shortest path length between p and q can be computed in $O(\log n)$ time and the path itself can be found in additional time proportional to the number of turns along it.

Our query algorithm for finding v^* works as follows.

First, we use the ray-shooting data structure to find in $O(\log n)$ time the first point q^* on the boundary of \mathcal{P} hit by ρ ; the edge of \mathcal{P} containing q^* is also known immediately from the ray-shooting query. If q^* is a vertex of \mathcal{P} , then $v^* = q^*$ and we are done; otherwise, let the end vertices of the edge of \mathcal{P} containing q^* be u_1 and u_2 (e.g., see Fig. 4). Let π_1 be the shortest path in \mathcal{P} from z to u_1 , and similarly, let π_2 be the shortest path from z to u_2 . Since z is visible to q^* on $\overline{u_1 u_2}$, the region bounded by π_1 , π_2 , and $\overline{u_1 u_2}$ is a funnel [11,12,17], with z as the apex (e.g., see Fig. 4). Recall that ρ is horizontally rightwards; one vertex of u_1 and u_2 must be below the line containing ρ . Without loss of generality, let u_1 be below the line containing ρ . Let v be the vertex on π_1 that is connected to z by a line segment on π_1 , i.e., \overline{zv} is the first edge of π_1 (e.g., see Fig. 4). Note that $v = u_1$ is possible, in which case π_1 is the line segment $\overline{zu_1}$. An easy observation is that the sought vertex v^* is exactly the vertex v . By using the two-point shortest path data structure [12] on z and u_1 , the vertex v can be easily found in $O(\log n)$ time because \overline{zv} is the first edge of π_1 .

Therefore, the sought vertex v^* for the ray-rotating query on ρ can be found in $O(\log n)$ time. The lemma thus follows. \square

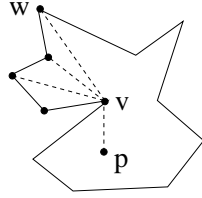


Fig. 5. Illustrating the principle child w of v in $T(p)$.

3 The First Data Structure

Our goal is to compute $Vis(s)$ for any query segment s . Again, let $s = \overline{ab}$. As discussed before, we need to resolve two issues. The first issue is to compute $Vis(a)$. For this, as discussed in [1], by using the ray-shooting data structure [5,6,11,15], with $O(n)$ time preprocessing, we can compute $Vis(a)$ in $O(|Vis(a)| \log n)$ time, where $|Vis(a)|$ is the size of $Vis(a)$. Note that it might be easier to compute $|Vis(a)|$ by using both the ray-shooting data structure and our ray-rotating data structure in Lemma 1.

The second issue is how to determine the next critical constraint of \mathcal{P} that will be crossed by the point p as p moves from a to b . Suppose at the moment we know $Vis(p)$ (initially, $Vis(p) = Vis(a)$). Let β be the critical constraint that is crossed next by p . To determine β , we first sketch an observation given in [1].

Denote by $T(p)$ the shortest path tree rooted at p , which is the union of the shortest paths in \mathcal{P} from p to all vertices of \mathcal{P} . A vertex of \mathcal{P} is in $Vis(p)$ if and only if it is a child of p in $T(p)$. For any child v of p in the tree $T(p)$, define the *principal child* of v to be the vertex w among the children of v in $T(p)$ such that the angle formed by the rays \overrightarrow{vw} and \overrightarrow{pv} is the smallest among all such angles (see Fig. 5). In other words, if we go from p to any child of v along the shortest path and we turn to the left (resp., right), then w is the first child of v that is hit by rotating counterclockwise (resp., clockwise) the line containing \overline{pv} around v .

To determine the next critical constraint β , the following observation was shown in [1]. Two children of p in $T(p)$ are *consecutive* if there is no other child of p between them in the cyclic order around p .

Observation 1 [1] *The next critical constraint β is defined by two vertices of \mathcal{P} that are either two consecutive children of p or one, say v , is a child of p and the other is the principal child of v .*

Based on Observation 1, Aronov *et al.* [1] maintained $T(p)$ as p moves and used the balanced triangulation of \mathcal{P} to determine the principal children. Their query algorithm takes $O(k \log^2 n)$ time, where $k = |Vis(s)|$, and the preprocessing time and space of their data structure [1] are $O(n^2 \log n)$ and $O(n^2)$, respectively.

Here, we take a different approach, although we still use Observation 1. Our data structure consists of the following: the ray-shooting data structure [5,6,11,15], the ray-rotating data structure in Lemma 1, and a priority queue Q .

We assume $Vis(a)$ has been computed. We use the ray-rotating data structure to determine the principal children of all children of p , as follows. First of all, since we already know $Vis(p)$ (initially $p = a$), we have all p 's children in $T(p)$, sorted cyclically around p . Note that we need not store the entire tree $T(p)$. Consider any child v of p in $T(p)$ (i.e., v is visible to p). Let w be the principal child of v that we are looking for. Consider the ray $\rho(v)$ originating from v with the direction from p to v . By the definition of principle children, w is the first vertex of \mathcal{P} visible to v that will be hit by the ray $\rho(v)$ if we rotate $\rho(v)$ around v along the direction that is consistent with the turning direction of the shortest paths from p to the children of v in $T(p)$. It is easy to see that once we know the above rotation direction, we can obtain w in $O(\log n)$ time by our ray-rotating data structure in Lemma 1.

To determine the above rotation direction, we only need to look at the relationship between the line containing $\rho(v)$ and the two edges of \mathcal{P} adjacent to v . Specifically, assume the line containing $\rho(v)$ has the same direction as $\rho(v)$. For example, if the two adjacent edges of v both lie to the left of this line (e.g., see Fig. 5), then we should rotate $\rho(v)$ counterclockwise to determine w . The other cases can be determined in a similar manner. In summary, we can obtain the principle child of v in $O(\log n)$ time. Initially, $p = a$ and we determine the principle children of all children of a in $T(a)$ in $O(|Vis(a)| \log n)$ time since a has $O(|Vis(a)|)$ children in $T(a)$.

We use the priority queue Q to store the critical constraints specified in Observation 1 that intersect the line segment s , where the key of each such critical constraint used in the priority queue Q is the position of its intersection with s . Initially when $p = a$, we compute the critical constraints defined by all pairs of consecutive children of p in $T(p)$. Similarly, for each child v of $T(p)$, we compute the critical constraint defined by v and its principal child. Note that the total number of these critical constraints is $O(|Vis(a)|)$. For each such critical constraint, we check whether it intersects s , which can be done in $O(\log n)$ time with the help of a ray-shooting query (we omit the details). If the critical constraint intersects s , we insert it into Q ; otherwise, we do nothing. Then, the first critical constraint in Q is the next critical constraint that p will cross as it moves. In general, after p crosses a critical constraint, p either sees one more vertex or sees one less vertex of \mathcal{P} . In either case, there are only a constant number of insertion or deletion operations on Q . Specifically, consider the case when p sees one more vertex u (and an adjacent edge of u). By the implementation given in [2], we can update the combinatorial representation of $Vis(p)$ in constant time (i.e., insert u and the adjacent edge to the appropriate positions of the cyclic list of $Vis(p)$). After this, u becomes a child of p in the new tree $T(p)$, and we can determine p 's two other children, say, u_1 and u_2 , which are cyclic neighbors of u , in constant time. Then, for u_1 , we check whether the critical constraint defined by u and u_1 intersects s , and if so, we insert it into Q . For u_2 , we do the same thing. Further, we compute the principal child of u in $T(p)$, in $O(\log n)$ time, by the approach discussed above. For the other case where p sees one less vertex after it crosses

the critical constraint, we perform similar processing. After p arrives at the other endpoint b of s , we obtain the combinatorial representation of $Vis(s)$.

We claim that the above algorithm takes $O(k \log n)$ time (with $k = |Vis(s)|$). Indeed, the initialization takes $O(|Vis(a)| \log n)$ time. Clearly, $|Vis(a)| = O(k)$ since each vertex of \mathcal{P} that is in $Vis(a)$ also appears in $Vis(s)$. If we consider every time when p crosses a critical constraint as an *event*, then each event takes $O(\log n)$ time. It has been shown in [1] that the total number of events as p moves from a to b is $O(k)$. Hence, the overall running time for computing $Vis(s)$ is $O(k \log n)$.

For the preprocessing, the ray-shooting data structure and the ray-rotating data structure both need $O(n)$ time and space to build. Further, in our query algorithm, the space used in the priority queue Q is always bounded by $O(k)$. We conclude this section with the following result.

Theorem 1. *For any simple polygon \mathcal{P} , a data structure can be built in $O(n)$ time and $O(n)$ space, such that the visibility polygon $Vis(s)$ can be computed in $O(|Vis(s)| \log n)$ time for any query line segment s in \mathcal{P} .*

4 The Second Data Structure

In general, the preprocessing of our second data structure is very similar to that in [2], and we make it faster by using better tools. Our improvement on the query algorithm is based on a number of new observations, e.g., a combinatorial bound of the “zone” of the line segment arrangements in simple polygons. For completeness, we first briefly discuss the approach in [2].

The preprocessing in [2] has several steps, whose running time is $O(n^3 \log n)$ and is dominated by the first two steps. The other steps together take $O(n^3)$ time. We show below that the first two steps can be implemented in $O(n^3)$ time.

The preprocessing in [2] first computes the visibility decomposition $VD(\mathcal{P})$ of \mathcal{P} . Although there may be $\Omega(n^2)$ critical constraints in \mathcal{P} , it has been shown [2] that any line segment in \mathcal{P} can intersect only $O(n)$ critical constraints, which implies that the size of $VD(\mathcal{P})$ is $O(n^3)$ instead of $O(n^4)$. All critical constraints of \mathcal{P} can be computed in $O(n^2)$ time, e.g., by the algorithm in [10]. After that, to compute $VD(\mathcal{P})$, we can use Chazelle and Edelsbrunner’s algorithm [4], which computes the planar subdivision induced by a set of m line segments in $O(m \log m + I)$ time, where I is the number of intersections of the line segments. In our problem, we have $O(n^2)$ critical constraints each of which is a line segment and the boundary of \mathcal{P} has n edges. Therefore, by using the algorithm in [4], we can compute $VD(\mathcal{P})$ in $O(n^3)$ time. Alternatively, an approach mentioned in [15] can also be used to compute $VD(\mathcal{P})$ in $O(n^3)$ time, and we omit the details.

The second step of the preprocessing in [2] is to build a planar point location data structure on $VD(\mathcal{P})$ in $O(n^3 \log n)$ time. By the approaches in [9] or [16], we can build such a point location data structure in $O(n^3)$ time.

The remaining steps of our preprocessing algorithm are the same as those in [2], which together take $O(n^3)$ time. Hence, the total preprocessing time is

$O(n^3)$. With the preprocessing, for each query point q in \mathcal{P} , we can compute the visibility polygon $Vis(q)$ of q in $O(|Vis(q)| + \log n)$ time.

For a query segment $s = \overline{ab}$, the query algorithm in [2] first computes $Vis(a)$. Then, again, let a point p move on s from a to b . The algorithm maintains $Vis(p)$ as p moves on s , initially with $Vis(p) = Vis(a)$. Again, whenever p crosses a critical constraint, the combinatorial representation of $Vis(p)$ changes. Unlike our first data structure in Section 3, here we have $VD(\mathcal{P})$ explicitly. Therefore, we can determine the next critical constraint in a much easier way. Specifically, the algorithm in [2] uses the following approach. Suppose p is currently in a cell of $VD(\mathcal{P})$; then the next critical constraint crossed by p must be on the boundary of that cell. Since each cell is convex, we can determine this critical constraint in $O(\log n)$ time. The algorithm stops when p arrives at b . The total running time of the query algorithm is $O(k \log n)$, where $k = |Vis(s)|$.

We propose a new and simpler query algorithm. We follow the previous query algorithmic scheme. The only difference is when we determine the next critical constraint that will be crossed by p , we simply check each edge on the boundary of the current cell that contains p , and the running time is linear in terms of the number of edges of the cell. Therefore, the total running time of finding all critical constraints crossed by p as it moves on s is proportional to the total number of edges on all faces of $VD(\mathcal{P})$ that intersect s , and we denote by $F(s)$ the set of such faces of $VD(\mathcal{P})$. Let $E(s)$ denote the set of edges of the faces in $F(s)$. Then the total time of finding all critical constraints crossed by p is $O(|E(s)|)$. Note that the time of the overall query algorithm is the sum of the time for computing $Vis(a)$ and the time for finding all critical constraints crossed by p . Since $Vis(a)$ can be found in $O(|Vis(a)| + \log n)$ time, the running time of the query algorithm is $O(\log n + |Vis(a)| + |E(s)|)$. Recall that $|Vis(a)| = O(k)$. In Lemma 2 below, we will prove $|E(s)| = O(k)$. Consequently, the query algorithm takes $O(\log n + k)$ time and Theorem 2 below thus follows.

Lemma 2. *The size of the set $E(s)$ is $O(k)$.*

Theorem 2. *For any simple polygon \mathcal{P} , we can build a data structure of size $O(n^3)$ in $O(n^3)$ time that can compute $Vis(s)$ in $O(|Vis(s)| + \log n)$ time for each query segment s in \mathcal{P} .*

4.1 Proving Lemma 2

It remains to prove Lemma 2. Note that each edge of $E(s)$ lies either on $\partial\mathcal{P}$ or on a critical constraint. We partition the set $E(s)$ into two subsets $E_1(s)$ and $E_2(s)$. For each edge of $E(s)$, if it lies on $\partial\mathcal{P}$, then it is in $E_1(s)$; otherwise, it is in $E_2(s)$. We will show that both $|E_1(s)| = O(k)$ and $|E_2(s)| = O(k)$ hold.

Denote by $C(s)$ the set of all critical constraints each of which contains at least one edge of $E(s)$.

Lemma 3. *The size of the set $C(s)$ is $O(k)$.*

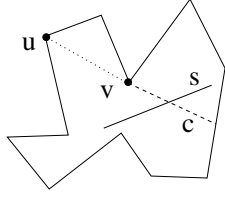


Fig. 6. Illustrating the case where c intersects s .

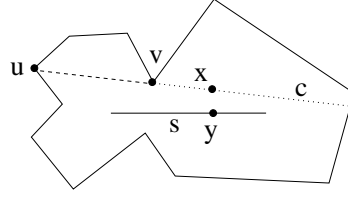


Fig. 7. Illustrating the case where c does not intersect s .

Proof: Denote by $V(s)$ the set of vertices of \mathcal{P} visible to s . Clearly, $|V(s)| \leq k$. Consider an arbitrary critical constraint $c \in C(s)$. To prove the lemma, we will charge c to a vertex of $V(s)$. We will show that each vertex of $V(s)$ will be charged at most a constant number of times, which will lead to the lemma.

Assume the defining vertex of c is u and the anchor vertex of c is v . By the definition of $C(s)$, c contains at least one edge of $E(s)$. Depending on whether c intersects s , there are two cases.

1. If c intersects s , then the defining vertex u of c must be visible to s (see Fig. 6). To see this, let q be the intersection of c and s . Hence, q is visible to v . One may consider that the visibility between q and u is blocked by v . Due to our assumption that each endpoint of s is not collinear with two vertices of \mathcal{P} , q is not an endpoint of s . Hence, there is always a point on s infinitely close to q that is visible to u , and thus u is visible to s . We charge c to its defining vertex u .
2. If c does not intersect s (see Fig. 7), then we show below that the anchor vertex v of c must be visible to s , and further, there are at most two critical constraints in $C(s)$ such that they do not intersect s and their anchor vertices are v . We will charge c to v .

We first prove that v is visible to s . Since c contains at least one edge of $E(s)$, there must be a face f of $VD(\mathcal{P})$ intersecting s and the boundary of f has an edge e contained in c . Let x be an arbitrary interior point of e and let y be an arbitrary point on s that is contained in f (see Fig. 7). Since f is convex, \overline{xy} is contained in f , i.e., x is visible to y and \overline{xy} does not intersect any other critical constraint of \mathcal{P} than c (at x). To prove y is visible to the vertex v , consider a point q on \overline{xy} moving from x to y . We claim that v is always visible to q as q moves. Indeed, initially q is at x , and v is visible to x because x is on the critical constraint c and v is the anchor vertex of c . Suppose to the contrary v is not visible to q at some moment as q moves. Then, at some moment, \overline{vq} must encounter a vertex of \mathcal{P} , say, w . In other words, w is on \overline{vq} . Then, the two vertices v and w define a critical constraint with v as the defining vertex and w as the anchor vertex, and the critical constraint intersects \overline{xy} at q . Note that this critical constraint is not c because v is the anchor vertex of c . Hence, we obtain a contradiction because c is the only critical constraint that intersects \overline{xy} . Therefore, we conclude that v is visible to y .

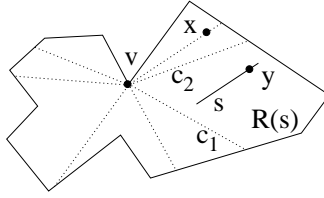


Fig. 8. Illustrating the critical constraints (the dotted line segments) with v as their anchor vertex that do not intersect s .

Next, we prove that there are at most two critical constraints in $C(s)$ such that they do not intersect s and their anchor vertices are v . Let C_v denote the set of all critical constraints each of which has v as its anchor vertex and does not intersect s . Our goal is to prove that C_v has at most two critical constraints in $C(s)$. Note that each critical constraint in C_v has v as an endpoint and its other endpoint is on $\partial\mathcal{P}$. Hence, the critical constraints of C_v partition \mathcal{P} into $|C_v| + 1$ interior-disjoint regions and one region contains s (see Fig. 8). Let $R(s)$ be the region containing s . Clearly, $R(s)$ has at most two critical constraints of C_v , say c_1 and c_2 , on its boundary. We claim that for any critical constraint $c' \in C_v \setminus \{c_1, c_2\}$, c' cannot contain an edge of $E(s)$. Indeed, assume to the contrary c' contains an edge of $E(s)$. Then, as discussed before, we can always find such two points x and y as in Fig. 7. Recall that \overline{xy} is in \mathcal{P} and \overline{xy} does not intersect any other critical constraint of \mathcal{P} than c' . Since c' is outside $R(s)$, $x \in c'$ is outside $R(s)$. However, due to $y \in s$ and $s \subset R(s)$, \overline{xy} must intersect either c_1 or c_2 , which contradicts with that \overline{xy} does not intersect any other critical constraint of \mathcal{P} than c' . Hence, c' cannot contain an edge of $E(s)$ and $c' \notin C(s)$. Therefore, we obtain that C_v has at most two critical constraints in $C(s)$.

According to our discussion above, in the first case (i.e., c intersects s), we charge c to its defining vertex u , which is in $V(s)$. In the second case (i.e., c does not intersect s), we charge c to its anchor vertex v , which is also in $V(s)$. An observation in [2] shows that for any line segment in \mathcal{P} , for any vertex u of \mathcal{P} , the line segment intersects at most two critical constraints with u as their defining vertex. Therefore, for any vertex u of \mathcal{P} , u can be charged at most twice as a defining vertex. On the other hand, we have shown that, as an anchor vertex, v has at most two critical constraints in $C(s)$ that do not intersect c . Therefore, for any vertex v of \mathcal{P} , v can be charged at most twice as an anchor vertex. Hence, any vertex in $V(s)$ can be charged at most four times, twice as an anchor vertex and twice as a defining vertex. In other words, $|C(s)| \leq 4 \cdot |V(s)| \leq 4 \cdot k$.

The lemma thus follows. \square

In the next lemma, we bound the size of the subset $E_1(s)$.

Lemma 4. *The size of the set $E_1(s)$ is $O(k)$.*

Proof: Denote by $V(s)$ the set of vertices of \mathcal{P} visible to s . Clearly, $|V(s)| \leq k$. Consider an edge e in $E_1(s)$. To prove the lemma, we will charge e either to a

vertex of $V(s)$ or to a critical constraint of $C(s)$. We will also show that each vertex of $V(s)$ will be charged at most twice and each critical constraint of $C(s)$ will be charged at most four times. Consequently, due to $|V(s)| \leq k$ and $|C(s)| = O(k)$ (by Lemma 3), the lemma follows.

By the definition of $E_1(s)$, e is on an edge of \mathcal{P} . If e has an endpoint that is a vertex of \mathcal{P} , say u , then clearly u is visible to s . We charge e to u . Otherwise, both endpoints of e are endpoints of some critical constraints, and we charge e to an arbitrary one of the two such critical constraints.

For each vertex of \mathcal{P} , it has two adjacent edges in \mathcal{P} , and therefore, it has at most two adjacent edges in $E_1(s)$. Hence, each vertex of $V(s)$ can be charged at most twice. On the other hand, each critical constraint has two endpoints, and each endpoint is adjacent to at most two edges in $E_1(s)$. Therefore, each critical constraint of $C(s)$ can be charged at most four times. \square

To prove Lemma 2, it remains to show $|E_2(s)| = O(k)$. To this end, we discuss a more general problem, in the following.

Assume we have a set S of line segments in \mathcal{P} such that the endpoints of each such segment are on $\partial\mathcal{P}$. Let \mathcal{A} be the arrangement formed by the line segments of S and the edges of $\partial\mathcal{P}$. For any line segment s in \mathcal{P} (the endpoints of s need not be on $\partial\mathcal{P}$), the *zone* of s is defined to be the set of all faces of \mathcal{A} that s intersects. Denote by $Z(s)$ the zone of s . For each edge of a face in \mathcal{A} , it either lies on a line segment of S or lies on $\partial\mathcal{P}$; if it is the former case, we call the edge an *S-edge*. We define the *complexity* of $Z(s)$ as the number of *S-edges* of the faces in $Z(s)$ (namely, the edges on $\partial\mathcal{P}$ are not considered), denoted by Λ . Our goal is to find a good upper bound for Λ . By using the zone theorem for the general line segment arrangement [8], we can easily obtain $\Lambda = O(|S|\alpha(|S|))$, where $\alpha(\cdot)$ is the functional inverse of Ackermann's function [13].

Denote by S_s the set of line segments in S that intersect $Z(s)$, i.e., each segment in S_s contains at least one *S-edge* of $Z(s)$. Let $m = |S_s|$ (note that $m \leq |S|$). By using the property that each segment in S has both endpoints on $\partial\mathcal{P}$, we show $\Lambda = O(m)$ in Theorem 3 below, which we call the *zone theorem*. The proof of Theorem 3 is given in Section 4.2.

Theorem 3. *The complexity of $Z(s)$ is $O(m)$.*

Now consider our original problem of proving $|E_2(s)| = O(k)$. By using the zone theorem, we have the following corollary.

Corollary 1. *The size of the set $E_2(s)$ is $O(k)$.*

Proof: The set $E_2(s)$ consists of all edges of $E(s)$ that lie on the critical constraints. Recall that each critical constraint is a line segment in \mathcal{P} with both endpoints on $\partial\mathcal{P}$. Consider the arrangement formed by all critical constraints of \mathcal{P} and $\partial\mathcal{P}$. The complexity of the zone $Z(s)$ of the query segment s in this arrangement is exactly $|E_2(s)|$. Let $C'(s)$ be the set of critical constraints of \mathcal{P} each of which contains at least one edge in $E_2(s)$. Then, by the zone theorem (Theorem 3), we have $|E_2(s)| = O(|C'(s)|)$. Note that $C'(s) \subseteq C(s)$. Due to

$|C(s)| = O(k)$ (Lemma 3), we have $|E_2(s)| = O(k)$. The corollary thus follows. \square

Lemma 4 and Corollary 1 together lead to Lemma 2.

4.2 Proving the Zone Theorem (i.e., Theorem 3)

This subsection is devoted entirely to proving the zone theorem, i.e., Theorem 3. All notations here are the same as defined before.

We partition the set S_s into two subsets: S_s^1 and S_s^2 . For each segment in S_s , if it does not intersect the interior of s , then it is in S_s^1 ; otherwise, it is in S_s^2 . Let $m_1 = |S_s^1|$ and $m_2 = |S_s^2|$. Hence, $m = m_1 + m_2$. Consider the arrangement formed by the line segments in S_s^1 and $\partial\mathcal{P}$. Since no segment in S_s^1 intersects the interior of s , s must be contained in a single face of the above arrangement and we denote by F_s that face. For each edge of F_s , if it lies on a segment of S , we also call it an S -edge. Note that the edges of F_s that are not S -edges are all on $\partial\mathcal{P}$.

Lemma 5. *The number of S -edges of the face F_s is $O(m_1)$; the shortest path in \mathcal{P} between any two points in F_s is contained in F_s .*

Proof: For each segment s' in S_s^1 , since both endpoints of s' are on $\partial\mathcal{P}$, s' partitions \mathcal{P} into two simple polygons and one of them contains s , which we denote by $\mathcal{P}(s')$. It is easy to see that the face F_s is the common intersection of $\mathcal{P}(s')$'s for all s' in S_s^1 . To prove the lemma, it is sufficient to show that each segment s' in S_s^1 has at most one (maximal) continuous portion on the boundary of F_s , as follows.

For any two points p and q in \mathcal{P} , denote by $\pi(p, q)$ the shortest path between p and q in \mathcal{P} . Note that since \mathcal{P} is a simple polygon, $\pi(p, q)$ is unique. We claim that for any two points p and q in the face F_s , $\pi(p, q)$ is contained in F_s . Indeed, suppose to the contrary $\pi(p, q)$ is not contained in F_s . Then, $\pi(p, q)$ must cross the boundary of F_s . Since $\pi(p, q)$ cannot cross the boundary of \mathcal{P} , $\pi(p, q)$ must cross an S -edge of F_s , and we assume s' is the segment in S_s^1 that contains such an S -edge. This implies that $\pi(p, q)$ is also not contained in the polygon $\mathcal{P}(s')$. Recall that the line segment s' partitions \mathcal{P} into two simple polygons and one of them is $\mathcal{P}(s')$. It is easy to show that for any two points in $\mathcal{P}(s')$, their shortest path in \mathcal{P} must be contained in $\mathcal{P}(s')$. Therefore, we obtain a contradiction. Hence, our above claim is true.

Now assume to the contrary that a segment s' in S_s^1 has two disjoint maximal continuous portions on the boundary of F_s . Let p and q be two points on these two portions of s' , respectively. Thus, both p and q are in F_s . Since these are two discontinuous portions of s' on the boundary of F_s , the line segment \overline{pq} is not contained in F_s . Since \overline{pq} is on s' , the shortest path $\pi(p, q)$ is \overline{pq} . But this means $\pi(p, q)$ is not contained in F_s , which incurs a contradiction with our previous claim that $\pi(p, q)$ must be contained in F_s . Hence, we obtain that each segment s' in S_s^1 has at most one continuous portion on the boundary of F_s , and consequently, the number of S -edges of the face F_s is $O(m_1)$. \square

Lemma 6 below shows a property of the face F_s .

Lemma 6. *For any line segment s' in \mathcal{P} with both endpoints on $\partial\mathcal{P}$, s' has at most one (maximal) continuous portion intersecting F_s ; consequently, s' intersects the interior of at most two edges of F_s .*

Proof: Assume to the contrary that s' has two disjoint maximal continuous portions intersecting F_s . Let p and q be two points on these two portions of s' , respectively. Thus, both p and q are in F_s . Clearly, the line segment \overline{pq} is not contained in F_s . Since \overline{pq} is on s' , \overline{pq} is the shortest path $\pi(p, q)$ between p and q in \mathcal{P} . But this means $\pi(p, q)$ is not contained in F_s , which incurs a contradiction with Lemma 5. Hence, the lemma holds. \square

For each S -edge of $Z(s)$, it lies either on a segment in S_s^1 or on a segment in S_s^2 ; we call it an S_s^1 -edge if it lies on a segment in S_s^1 and an S_s^2 -edge otherwise. Due to $m = m_1 + m_2$, our zone theorem is an immediate consequence of Lemma 7 below. Note that we can obtain the zone $Z(s)$ of s by adding the segments of S_s^2 to F_s . To prove Lemma 7, we use induction on m_2 , i.e., $|S_s^2|$. The approach is very similar to that in [7] used for line arrangements. Here, although we have line segments instead of lines, the property that each line segment has both endpoints on $\partial\mathcal{P}$ makes the approach in [7] applicable with some modifications.

Lemma 7. *There are $O(m_2)$ S_s^2 -edges and $O(m_1 + m_2)$ S_s^1 -edges in the zone $Z(s)$.*

Proof: Without loss of generality, assume the segment s is horizontal. It is easy to see that each S_s^1 -edge bounds one face of $Z(s)$ and each S_s^2 -edge bounds two faces of $Z(s)$ (one lies on its right and the other lies on its left). For each S_s^2 -edge, we say it is a *left bounding S_s^2 -edge* for the face lying on the right of it and a *right bounding S_s^2 -edge* for the face lying on the left of it. Below we will prove that the number of left bounding S_s^2 -edges of the faces in $Z(s)$ is $O(m_2)$. Analogously, the number of right bounding S_s^2 -edges of the faces in $Z(s)$ is also $O(m_2)$. In addition, we will also show that the number of S_s^1 -edges of $Z(s)$ is $O(m_1 + m_2) = O(m)$.

Our proof is by induction on m_2 . Consider the base case with $m_2 = 1$. Denote by s' the only segment in S_s^2 . Note that the face F_s has no S_s^2 -edges on its boundary and has $O(m_1)$ S_s^1 -edges by Lemma 5. In light of Lemma 6, s' has at most one maximal continuous portion intersecting the face F_s and s' intersects the interior of at most two S_s^1 -edges of F_s . Therefore, after we add s' to F_s , the number of S_s^1 -edges of $Z(s)$ increases by at most two and the number of left bounding S_s^2 -edges increases by at most one.

Consider the general case of $m_2 \geq 1$. Let s' be the segment in S_s^2 that intersects s at the rightmost position among all segments in S_s^2 . We first consider the case when this segment s' is unique. By induction, the zone of s has $c \cdot (m_2 - 1)$ left bounding S_s^2 -edges and $c \cdot (m_1 + m_2 - 1)$ S_s^1 -edges, for some constant c , without considering the segment s' . Now consider adding s' . First, by Lemma 6, s' has at most one maximal continuous portion intersecting the face F_s and s' intersects the interior of at most two S_s^1 -edges of the zone $Z(s)$; therefore, the number of S_s^1 -edges increases by at most 2. Second, the number of left bounding S_s^2 -edges increases in two ways: there are new left bounding S_s^2 -edges on s' and there are

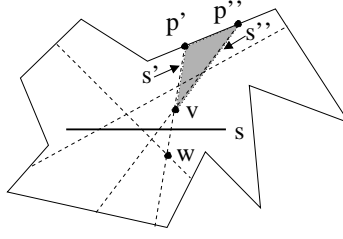


Fig. 9. The shaded region is $R(v)$, which is not in the zone of s .

old left bounding S_s^2 -edges that are split by s' . Let v be the first intersection point of s' with another segment in S_s^2 above s , and let w be the first intersection point of s' with another segment in S_s^2 below s (e.g., see Fig. 9). We assume both v and w exist since otherwise the analysis is even simpler. The segment \overline{vw} on s' becomes a new left bounding S_s^2 -edge. In addition, s' splits a left bounding S_s^2 -edge at v and at w , respectively. Hence, the number of the left bounding S_s^2 -edges increases by three. We claim that there is no other increase for the number of left bounding S_s^2 -edges.

Indeed, consider the part of s' above v . Let s'' be the segment in S_s^2 that intersects s' at v . Let p' be the endpoint of s' above v and p'' be the endpoint of s'' above v . Note that both p' and p'' are on $\partial\mathcal{P}$. Consider the region $R(v)$ above v enclosed by $\overline{vp'}$, $\overline{vp''}$, and the portion of $\partial\mathcal{P}$ between p' and p'' (e.g., see Fig. 9). Clearly, the region $R(v)$ is not in the zone of s . Further, $R(v)$ lies on the right of $\overline{vp'}$, and thus $\overline{vp'}$ cannot contribute any left bounding S_s^2 -edges to $Z(s)$. In addition, if a left bounding S_s^2 -edge e that was in the zone $Z(s)$ (before s' is added) is intersected by s' somewhere above v , then the part of e to the right of s' (i.e., the part of e in the region $R(v)$) is not in the zone $Z(s)$ any more after s' is added. Hence, there is no increase in the number of left bounding S_s^2 -edges due to such an intersection.

In a similar way, we can show that the part of s below w does not increase the number of left bounding S_s^2 -edges in the zone $Z(s)$. Therefore, after s' is added, the total increase of the number of left bounding S_s^2 -edges is at most three.

We discuss above the case when s' is the only segment in S_s^2 through the rightmost intersection on s . If there is more than one such segment, then we take an arbitrary such segment as s' . By a similar analysis as that above and that in [7], we can show that the total increase in the number of left bounding S_s^2 -edges is at most five. We omit the details.

We conclude that there are $O(m_2)$ S_s^2 -edges and $O(m_1 + m_2)$ S_s^1 -edges in the zone $Z(s)$. The lemma thus follows. \square

5 Conclusions

In this paper, we propose two new data structures for the weak visibility query problem on a simple polygon, which improve upon the previous work [1,2,3].

Some results (e.g., the ray-rotating data structure and the zone theorem) may be of independent interest.

For the $O(k \log n)$ time queries, our first data structure is clearly optimal. For the $O(k + \log n)$ time query, however, an open question is whether a data structure of sub-cubic preprocessing time and space is possible.

References

1. B. Aronov, L.J. Guibas, M. Teichmann, and L. Zhang. Visibility queries and maintenance in simple polygons. *Discrete and Computational Geometry*, 27(4):461–483, 2002.
2. P. Bose, A. Lubiw, and J.I. Munro. Efficient visibility queries in simple polygons. *Computational Geometry: Theory and Applications*, 23(3):313–335, 2002.
3. M.N. Bygi and M. Ghodsi. Weak visibility queries in simple polygons. In *Proc. of the 23rd Canadian Conference on Computational Geometry*, 2011.
4. B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.
5. B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, J. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12(1):54–68, 1994.
6. B. Chazelle and L. Guibas. Visibility and intersection problems in plane geometry. *Discrete and Computational Geometry*, 4:551–589, 1989.
7. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry — Algorithms and Applications*. Springer-Verlag, Berlin, 3rd edition, 2008.
8. H. Edelsbrunner, L. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir. Arrangements of curves in the plane topology, combinatorics, and algorithms. *Theoretical Computer Science*, 92(2):319–336, 1992.
9. H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.
10. S.K. Ghosh and D.M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM Journal on Computing*, 20(5):888–910, 1991.
11. L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1-4):209–233, 1987.
12. L.J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, 1989.
13. S. Hart and M. Sharir. Nonlinearity of Davenport–Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6(2):151–177, 1986.
14. J. Hershberger. An optimal visibility graph algorithm for triangulated simple polygons. *Algorithmica*, 4:141–155, 1989.
15. J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18(3):403–431, 1995.
16. D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.
17. D.T. Lee and F.P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.